Le logger Python

Génération de fichiers de log

Par GALODE Alexandre

Date de publication : 24 mars 2015

Dernière mise à jour : 24 mars 2015

Quel que soit le type de programme écrit, script ou autre, il est forcément intéressant à un moment donné de tracer ce qui se passe durant l'exécution. Cela permet de pouvoir analyser une exécution ayant entraîné une erreur par exemple.

Par principe ce genre d'informations est placé dans un fichier de log, extension « .log », également appelé journal en français.

Je vous propose ici de voir comment faire avec Python.

Commentez

I - Introduction

Tout comme dans beaucoup d'autres langages, Python possède son propre module de log, le module « logging », afin de suivre le fonctionnement. Ce que l'on désire lors de l'exécution d'un programme. Installé par défaut avec Python, il est très simple d'emploi.

Ce module étant relativement complet, nous allons ici nous concentrer sur l'essentiel. Pour plus de fonctionnalités, je vous invite à regarder la **documentation officielle**.

II - Les bases

II-A - Principe de fonctionnement

Un logger repose sur un principe de fonctionnement très simple : à la demande, certaines informations sont stockées dans un fichier texte, notre fichier de log.

Pour ce faire, on définit généralement des niveaux de criticité qui indiquent l'importance du message. Il peut s'agir d'un simple message informatif (ex. : communication arduino établie) ou bien d'un message très important (ex. : l'application s'est arrêtée subitement. Erreur d'argument dans la méthode serial_com_arduino).

Le but est tout simplement de tracer l'activité de notre programme afin de pouvoir analyser son exécution, en général en cas d'anomalies.

II-B - Les différents niveaux de log

Les logs sont répartis en différents niveaux. Chaque niveau correspond à une importance donnée. Par défaut cinq niveaux sont prédéfinis. S'il vous est possible de définir vos propres niveaux, chose qui ne sera pas vue dans ce tutoriel, les cinq niveaux prédéfinis seront cependant suffisants dans la plupart des cas.

Niveau défini	Niveau pris en compte					Description
	C	E	W	I	D	
CRITICAL (C)						Désigne en général une erreur ayant entraîné le plantage de l'application
ERROR (E)						Désigne une erreur sévère, mais n'ayant pas entraîné le crash du programme.
WARNING (W)						Désigne un élément/ événement qui pourrait

						avoir son importance détection d'instabilité, seuil atteint…
INFO (I)	✔	✔	✔	✔		Niveau d'information standard. Permet de tracer les événements classiques. Par exemple, le lancement d'une fonction, transfert d'un fichier…
DEBUG (D)	✔	✔	✔	✔	✔	Ne sert que pour le débogage. Sert à tracer des informations détaillées : valeur de variables, trame de communication…

Comme nous pouvons le constater, lorsqu'on définit un niveau, correspondant en réalité à un seuil de sensibilité, nous loggons ce niveau ainsi que tous les niveaux supérieurs de criticité. Par exemple, si nous définissons la sensibilité de notre log sur « warning », nous loggerons l'ensemble des informations de type « warning », « error » et « critical ».

Il est assez rare de voir cette sensibilité codée en dur. En général il s'agit plutôt d'un paramètre en base de données ou dans un fichier de configuration, ce qui permet ainsi aux équipes support de pouvoir aisément changer ce seuil de sensibilité en vue de faire du débogage chez le client.

II-C - Création d'un logger basique

```
1. import logging
2.
3. logging.basicConfig(filename='test_log.log',level=logging.DEBUG,\
4.       format='%(asctime)s -- %(name)s -- %(levelname)s -- %(message)s')
5. logging.debug('Debug error')
6. logging.info('INFO ERROR')
7. logging.warning('Warning Error %s: %s', '01234', 'Erreur Oracle')
8. logging.error('error message')
9. logging.critical('critical error')
```

Ligne 1, nous importons notre module, rien de sorcier.

Ligne 3, nous définissons notre fichier de log. Nous passons tout d'abord le filename, en chemin relatif (ici au même niveau que notre script) ou absolu. Ensuite, nous définissons le niveau de sensibilité, puis enfin le format de sortie.

Ligne 5, 6, 7, 8 et 9, vous voyez comment saisir des entrées dans le fichier de log. C'est au développeur de définir l'importance de l'information. Le seul argument à passer est alors le message à écrire, une simple chaîne de caractères.

Je vous invite maintenant à copier ce code dans un fichier, à l'exécuter, et à regarder le log généré.

Le logger est en mode append. Le fichier n'est donc jamais purgé avant d'être renseigné. Tenez-en compte lors de vos essais.

II-D - Format de sortie

```
1. format='%(asctime)s -- %(name)s -- %(levelname)s -- %(message)s'
```

Le format de sortie est entièrement paramétrable. Les caractères « -- » correspondent à un choix personnel. À vous de définir ce que vous désirez. Les attributs du format sont issus du tableau ci-dessous :

Attribut formaté	Description
%(asctime)s	Date au format « AAAA-MM-JJ HH:MM:SS,xxx ». Remarquons que nous disposons d'une précision à la milliseconde
%(created)f	Idem précédent, mais avec une date en timestamp (utilisation de time.time())
%(filename)s	Nom du fichier ayant écrit dans le log
%(funcName)s	Nom de la fonction contenant l'appel à l'écriture dans le log
%(levelname)s	Niveau du message
%(levelno)s	Numérotation logique du niveau du message (C:50, E:40, W:30, I:20, D:10)
%(lineno)d	Ligne où trouver l'appel à écriture dans le log. Relatif au fichier d'appel
%(module)s	Nom du module ayant appelé l'écriture dans le log
%(msecs)d	Temps d'exécution depuis le lancement du programme en millisecondes
%(message)s	Le message à logger
%(name)s	Le nom de l'utilisateur courant
%(pathname)s	Chemin absolu du fichier ayant appelé l'écriture
%(process)d	Le numéro du process courant
%(processName)s	Le nom du process courant
%(thread)d	L'ID du thread courant
%(threadName)s	Le nom du thread courant

II-E - Parser notre fichier de log

Comme nous l'avons vu précédemment, nos fichiers de logs respectent toujours un format que nous avons prédéfini.

De fait, il peut être relativement aisé d'analyser un tel fichier. Il suffit de lire le fichier ligne par ligne, puis de fractionner le contenu de cette ligne avec le caractère de séparation choisi (dans notre exemple précédent, « -- ».

Il en résulte alors une liste où chaque fraction correspond à un élément précis dont nous connaissons le rôle et la signification.

```
1. with open("test_log.log", "r") as log_file:
```

```
2.      for line in log_file.readlines():
3.          tmp = line.split("--")
4.          print("time: %s, user: %s, level: %s, message: %s" % (tmp[0], tmp[1], tmp[2], tmp[3]))
```

Ce code bien que très basique montre comment analyser le fichier de log créé plus tôt.

Ici notre analyseur fonctionne bien, car notre caractère de séparation servant au split ne se trouve jamais dans les messages. Il faut toujours veiller à ce détail pour éviter tout problème lors de cette étape.

III - Allons plus loin

En réalité, la méthode vue plus tôt masquait ce que nous allons faire maintenant. Il faut savoir que le fonctionnement d'un logger n'est pas aussi simple que ce que nous avons pu voir jusqu'à présent.

Dans les faits, il existe un objet logger et un objet handler. Le logger peut s'apparenter, à notre niveau actuel de compréhension, à un tampon filtrant. On lui passe un message à mettre en log, et il s'assure que le niveau de criticité correspond bien à ce qu'il a le droit de laisser passer.

Une fois cette étape franchie, il passe l'info à un handler, un gestionnaire, dont le but est de rediriger les messages vers un endroit désigné par le type de handler. Parmi ces types, nous avons la console (StreamHandler), les fichiers (FileHandler, WatchedFileHandler, RotatingFileHandler, TimeRotatingFileHandler), les mails (SmtpHandler), le réseau (SocketHandler), le web (HTTPHandler)…

C'est le handler qui définit le format du message de sortie (via un objet Formatter). Il peut, lui aussi, utiliser un seuil de sensibilité, et donc filtrer son contenu.

Le résultat de cette structure est que nous pouvons gérer de manière très fine la redirection de chaque information logger.

À partir de maintenant, nous utiliserons cette forme complète de logging.

Pour ceux qui voudraient aller encore plus loin, il existe des objets « Filter » qui permettent de rajouter une couche supplémentaire de filtrage à la fois sur les loggers et les handlers. Je vous renvoie alors vers la documentation officielle.

III-A - Séparation de logs

III-A-1 - Utilité

La séparation de logs est quelque chose de commun en informatique. Il ne s'agit ni plus ni moins que d'isoler les différents niveaux dans des fichiers distincts. En sortie nous aurons alors un fichier critical.log, error.log, warning.log…

Le but est de compartimenter les messages afin d'en faciliter le traitement. Cette solution possède néanmoins un inconvénient de taille : si vous désirez mieux comprendre le déroulement des opérations ayant entraîné l'anomalie, vous êtes obligé de jouer avec plusieurs logs au lieu d'un seul.

En contrepartie, si vous ne désirez analyser qu'un seul niveau en particulier, alors cela vous facilitera la tâche.

III-A-2 - Fonctionnement

Nous allons aborder ici la forme complète de logging pour la première fois. Avant toute chose, le code.

```
1. import logging
2.
3.
4. formatter = logging.Formatter("%(asctime)s -- %(name)s -- %(levelname)s -- %(message)s")
5.
6. handler_critic = logging.FileHandler("critic.log", mode="a", encoding="utf-8")
7. handler_info = logging.FileHandler("info.log", mode="a", encoding="utf-8")
8.
9. handler_critic.setFormatter(formatter)
10. handler_info.setFormatter(formatter)
11.
12. handler_info.setLevel(logging.INFO)
13. handler_critic.setLevel(logging.CRITICAL)
14.
15. logger = logging.getLogger("nom_programme")
16. logger.setLevel(logging.INFO)
17. logger.addHandler(handler_critic)
18. logger.addHandler(handler_info)
19.
20. logger.debug('Debug error')
21. logger.info('INFO ERROR')
22. logger.critical('INFO ERROR2')
```

Tout d'abord, l'import qui ne change pas, et se passe de tout commentaire.

Vient ensuite, l'objet « Formatter ». Cet objet permet de définir le format désiré pour les messages. Nous reprenons le même format que dans notre exemple précédent, et ce pour tous les logs de sortie.

Nous créons ensuite deux handlers, un par fichier, pour le log critic, et un pour le log de type INFO. Nous leur passons le nom du fichier de sortie, le mode d'écriture et le type d'encodage à utiliser.

Ensuite, nous leur précisons le format des messages à utiliser. Dans notre cas il s'agira de l'objet Formatter précédemment créé.

Après cela, nous fixons leur niveau respectif de sensibilité.

Puis nous créons un objet de type logger en lui assignant un nom, un niveau minimal de sensibilité, et en le liant aux deux handlers précédemment créés.

Pour rappel, les handlers correspondent à la sortie des logs, et le logger à l'entrée, côté programme.

Enfin, les trois dernières lignes permettent de transmettre des messages de logs.

Il est impératif de donner un nom à chaque objet logger. En cas de champ vide, chaque appel à « getLogger » vous renverra le même objet. Faites l'essai pour constater par vous-même le résultat.

Copiez-collez ce code et exécutez-le. Analysons les fichiers de sortie.

Tout d'abord, nous remarquerons l'existence effective de deux fichiers, nommés critic.log et info.log.

Ensuite, nous constaterons que le code nous a permis de choisir dans quel fichier écrire telle ou telle info, en indiquant tout simplement dans quel logger nous désirions écrire.

Enfin, nous pouvons voir que tout niveau inférieur au seuil défini n'est toujours pas loggé, mais que les niveaux supérieurs le sont bien.

Nous avons ainsi effectué simplement une séparation de log en disposant d'un point d'entrée (logger) unique.

III-B - Rotation de logs

III-B-1 - Utilité

La rotation de logs est quelque chose de très utile et très usité. Il tombe sous le sens de chacun de nous que lorsqu'un fichier de log, un fichier texte rappelons-le, devient trop gros, il peut vite devenir difficile à ouvrir, exploiter, communiquer à un tiers…

Dans un tel cas, le plus simple nous dirons-nous est de commencer un nouveau fichier. Cela est effectivement une solution. Et c'est justement pour l'automatiser qu'existe la rotation de log. Quand notre fichier aura atteint une certaine taille, ou après un certain temps écoulé, notre log sera renommé avec un numéro en fin de nom, et un nouveau fichier sera alors créé.

III-B-2 - Fonctionnement

```
1. import logging
2. from logging.handlers import RotatingFileHandler
3.
4.
5. formatter_debug = logging.Formatter("%(asctime)s -- %(name)s -- %(levelname)s -- %(message)s")
6. formatter_info = logging.Formatter("%(asctime)s -- %(name)s -- %(levelname)s -- %(message)s")
7.
8. logger_debug = logging.getLogger("debug_log")
9. handler_debug = logging.handlers.RotatingFileHandler("debug.log", mode="a", maxBytes= 100,
 backupCount= 1 , encoding="utf-8")
10. handler_debug.setFormatter(formatter_debug)
11. logger_debug.setLevel(logging.DEBUG)
12. logger_debug.addHandler(handler_debug)
13.
14. logger_info = logging.getLogger("info_log")
15. handler_info = logging.handlers.RotatingFileHandler("info.log", mode="a", maxBytes= 10,
 backupCount= 1,encoding="utf-8")
16. handler_info.setFormatter(formatter_info)
17. logger_info.setLevel(logging.INFO)
18. logger_info.addHandler(handler_info)
19.
20. logger_debug.debug('Debug error')
21. logger_info.debug('DEBUG ERROR')
22. logger_info.info('INFO ERROR')
23. logger_info.critical('INFO ERROR2')
```

Nous allons ici nous concentrer sur les modifications. Tout d'abord l'import. En effet nous devons importer les nouveaux handlers utilisés directement.

Ensuite à la création des handlers, nous précisons notre nouvel handler en lui passant des arguments : nom, mode, taille maximale (maxBytes), valeur de départ pour la numérotation (backupcount), encodage utilisé.

Nous noterons ici la valeur faible de la taille maximale du fichier. Elle est réglée exprès pour que vous puissiez constater le fonctionnement. Nous noterons aussi que la découpe s'effectue en termes de lignes. Ainsi une ligne de log sera entièrement renseignée et seulement après, le log sera sauvegardé.

```
1. import logging
2. from logging.handlers import TimedRotatingFileHandler
3. import time
4.
```

```
5.
6. formatter_debug = logging.Formatter("%(asctime)s -- %(name)s -- %(levelname)s -- %(message)s")
7. formatter_info = logging.Formatter("%(asctime)s -- %(name)s -- %(levelname)s -- %(message)s")
8.
9. logger_debug = logging.getLogger("debug_log")
10. handler_debug = logging.handlers.TimedRotatingFileHandler("debug.log", when="s", interval=5,
 encoding="utf-8")
11. handler_debug.setFormatter(formatter_debug)
12. logger_debug.setLevel(logging.DEBUG)
13. logger_debug.addHandler(handler_debug)
14.
15. logger_info = logging.getLogger("info_log")
16. handler_info = logging.handlers.TimedRotatingFileHandler("info.log", when="s", interval=5,
 encoding="utf-8")
17. handler_info.setFormatter(formatter_info)
18. logger_info.setLevel(logging.INFO)
19. logger_info.addHandler(handler_info)
20.
21. logger_debug.debug('Debug error')
22. logger_info.debug('DEBUG ERROR')
23. logger_info.info('INFO ERROR')
24. time.sleep(7)
25. logger_info.critical('INFO ERROR2')
```

Ici nous utilisons un handler qui permet de sauvegarder les logs non pas quand ils ont atteint une certaine taille, comme le précédent, mais au bout d'un certain temps.

Les deux arguments dédiés sont « when » qui permet de stipuler l'unité d'« interval ». When peut être au choix : S, M, H, D, W0 (pour lundi) à W6 (pour dimanche), midnight (00:00:00).

Dans notre exemple, un nouveau log est créé toutes les cinq secondes.

Si aucun log n'existe encore, un backup de log vide sera alors créé. Cela est normal.

IV - Conclusion

Ainsi, comme nous venons de le voir, que l'on utilise le module logging en mode basique ou avancé, Python permet de tracer très simplement l'activité d'un programme.

De plus, la distinction de différents niveaux de log permet d'affiner encore plus le travail et de filtrer au mieux la sortie de log en fonction du public visé.

Puissant, simple, et entièrement paramétrable, « logging » fait vraiment partie des modules basiques de Python à ne pas oublier lors d'une nouvelle création.

V - Remerciements

- **LittleWhite**
- **gorgonite**
- **ClaudeLELOUP**